An efficient management of correlation sets with broadcast

Jacopo Mauro[1,2], Maurizio Gabbrielli[1,2], Claudio Guidi[4], and Fabrizio Montesi[3]

[1] Department of Computer Science, University of Bologna, Italy.
[2] Lab. Focus, INRIA, Bologna, Italy
gabbri | jmauro@cs.unibo.it
[3] IT University of Copenhagen, Denmark
fabr@itu.dk
[4] italianaSoftware srl, Imola, Italy
cguidi@italianasoftware.com

Abstract. A fundamental aspect which affects the efficiency and the performance of Service-Oriented Architectures is the mechanism which allows to manage sessions and, in particular, to assign incoming messages to the correct sessions (also known as service instances). A relevant mechanism for solving this problem, first introduced by BPEL and then used in other languages (e.g. Jolie) is that one based on *correlation sets*. The BPEL and Jolie languages are currently allowing the use of messages whose target is only one session. However there are a lot of scenarios where being able to send a broadcast message to more than one session could be useful. Supporting such a broadcast primitive means to allow correlation sets which can contain unspecified variables and this can be very inefficient, since usual implementations in terms of hash tables cannot be used in this case.
In this paper we propose a data structure, based on radix trees and an algorithm for managing a correlation mechanism that supports the broadcast primitive, without degrading the performances.

1 Introduction

Service-Oriented Computing (SOC) is a paradigm for programming distributed applications by means of the composition of services. Services are autonomous, self-descriptive computational entities that can be dynamically discovered and composed in order to build more complex functionalities. The resulting systems, called Service-Oriented Architectures (SOA), have a wide diffusion; as of today the most prominent technology in this context consist of Web Services, a set of open specifications that focuses on interoperability and compatibility with existing infrastructures. This is mainly obtained through the adoption of the XML document format and by using HTTP as the underlying transport protocol for communications.
In a SOA services are loosely coupled, i.e. they stress a minimality on the dependencies that each service has w.r.t. the others, and can be stateful; this last

point is the case of orchestrators which maintain a state for each created session. Usually, in a stateful service a session is created at the first client invocation. But, differently from the object-oriented approach, SOC does not guarantee references for identifying the new session. Thus a fundamental aspect which affects the efficiency and the performance of SOAs is the mechanism which allows to manage sessions. In fact, in a typical pattern of interaction, a service may manage many different sessions, corresponding to different clients. Since communications are usually supported with stateless protocols (e.g. SOAP on HTTP), when a service receives a message from a client C the system must be able to identify which is the session corresponding to C and that, therefore, must receive the message. In other words, sessions usually need to be accessed only by those invokers (messages) which hold some specific rights.

A relevant mechanism for solving this problem, first introduced by BPEL [1] and then used in JOLIE [8, 9], COWS [6] and in other languages, is that based on *correlation sets*. Intuitively a correlation set is a set of variables whose values allow to distinguish sessions initiated by different clients. More precisely, both the sessions and the incoming messages contain some specific "correlation values" defining the variables in the correlation set. When a message m arrives it is routed to the session which has the same values as m for the correlation variables.

As a simple example of correlation set consider the case of a service S used for buying goods. Suppose that S handles all the communication of a specific customer using a unique session, while different customers have different sessions. Assuming that a customer is uniquely determined by her name and surname we can use a correlation set consisting of the two variables $name$ and $surname$ for determining the customer's session. Now let us suppose that S can receive the following three types of messages (with the obvious meaning): $buy(name, surname, product\_id)$; $delete\_order(name, surname, product\_id)$; $pay(name, surname, product\_id, credit\_card\_info)$. When a customer, say John Smith, wants to buy product 1 he can send a message of the form $buy(John, Smith, 1)$. When this message is received the service checks whether there is a session that correlates with it, i.e. whether there exists a session whose variables $name$ and $surname$ are respectively instantiated to the values $John$ and $Smith$. If this is the case message m is assigned to such session. On the other hand, if John Smith is a new customer and no session correlates with m then the message is not delivered (note however, that in this case a new session could be created which correlates with the message, see for example [2, 1, 5]).

The BPEL and Jolie languages are currently allowing the use of messages whose target is only one session. However there are a lot of scenarios where being able to send a broadcast message to more than one session could be useful. Let's consider for instance a cloud environment where every user can start, control and terminate a virtual machine on the cloud (a framework similar for instance to Amazon EC2). Let's suppose that we would like a unique entry point to this system and this entry point is a service that can receive and send messages to the users and the administrators of the cloud. We could consider to have a session

for every virtual machine and control the virtual machine through this session. The key to identify a session can be the union of the following fields:

- the name, surname and date of birth of the user (we assume that these values univocally determine the user);
- the kind of virtualized operating system (i.e Ubuntu, Windows, ...);
- the version of the operating system;
- the priority of the virtual machine (high, medium, low).

Having this key a user (say John Smith born on the 1st of Jan 1970) can start a Windows 7 machine with low priority sending for instance a message like *start(John,Smith,19700101,windows,7,low)*. Later he can control and terminate the session (and therefore the virtual machine) simply sending messages like *execute* or *terminate* specifying every time all the fields of the key.

On the other hand suppose now that an administrator wants to apply a patch to all the Windows virtual machines. Without a broadcast primitive he/she should retrieve all the keys of sessions controlling a Windows machine and later send them the message that triggers the application of the patch. For the programmer point of view this usually involves the definition of a session or service that keeps the log of all the sessions. This session/service often slows down the performances due to the creation or deletion of new sessions. On the other hand having a broadcast primitive an administrator could send:

- a message like *get_location()* that will be sent to every session for asking to the session which hardware machine is used to run the virtual machine;
- a message like *patch(operating_system, operating_system_version, ...)* to patch all the virtual machines with a certain operating system and version;
- a messages like *terminate(name, surname, birthday_date)* that can terminate all the virtual machines belonging to a user;
- messages like *stop(priority)* or *stop(operating_system, priority)* can be used to stop every virtual machine having a specific priority or operating system + priority.

These are only few examples of the use of broadcast primitives. Another important application for these messages is for the implementation of a publish/subscribe pattern: This is a messaging pattern where senders (publishers) of messages do not send the messages directly to specific receivers (subscribers). The messages are instead divided into classes and the subscribers subscribe for the reception of messages of a given class. The system is responsible for sending every message belonging to a certain class to every subscriber that has subscribed for that class. Publisher may not know who are the subscribers and vice versa.
This pattern can be easily implemented using broadcast and a service having a correlation set that contains the class identifier. Whenever a subscriber subscribes for a class, a new session responsible for the forwarding of the message is created. The publisher now can send a broadcast message specifying in the message its class. The correlation mechanism will check this value and route the

message to every session that has subscribed for that class. The session can later forward the message to the real subscriber.

The aim of this paper is to present a data structure and an implementation of the correlation mechanism that supports the broadcast primitive without degrading the performances of the correlation of normal messages.
The operations that a correlation mechanism has to support can be seen as the select, insert and delete operations of a relational database, where every tuple of the relation is a session. The correlation set is a key of a relation. When a normal message arrives it always contains a key that determine the target session. In the database analogy the correlation operation is then a "select" operation, and in the case of normal messages the (complete) key is used to retrieve the target session. On the contrary, a broadcast message specifies only part of the key, indeed its target is potentially a set of sessions. Continuing in the database analogy, the broadcast operation can be efficiently implemented by adding an index for every type of broadcast messages. However, since increasing the number of indexes decrease the performances of the insert and delete queries (i.e. creation and deletion of sessions), the less indexes we have the better it is. We will then define a solution that uses the minimal number of indexes needed to correlate the messages to the right sessions. The indexes will be implemented using radix trees.

We would like to underline that in this work we have taken as a starting point the correlation mechanism of Jolie. We made this choice because we find that Jolie correlation mechanism is more flexible than the BPEL one. For instance Jolie correlation variables are normal variables and not a late-bound constant like in BPEL. While in BPEL the values of a correlation set are defined only by a specially marked send or receive message and once defined they can not change, in Jolie the programmer can decide to instantiate or change the values of a correlation set at run time. In BPEL all the fields (correlation proprieties or correlation tokens) of a message key should be always defined. Jolie instead allows partially defined keys. This flexibility comes with a price: the implementation of the search of a correlating session is linear w.r.t. the number of session while in BPEL it is constant (usually hash table are used).

The correlation mechanism can be seen as a special case of the well know content-based publish/subscribe mechanism [11]. Indeed the correlation mechanism can be seen as a simpler content-based publish/subscribe mechanism where messages are notifications, sessions are subscriptions and correlation variables are attributes. The correlation mechanism exploits however two constraints that usually a content-based publish/subscribe mechanism does not have. In correlation, few attributes need to be considered and only equality predicates are used to compare the attributes. Hence, this work could be considered as an improvement over publish/subscribe algorithms such as [4, 3] for scenarios where the previous two constraints hold.

After having provided some background in Section 2 we explain the idea of the algorithm in Section 3. In Section 4 we show how the data structure is created and used, while in Section 5 we prove the correctness of the algorithm

and we perform some complexity analysis. Finally Section 6 concludes describing some future work.

2 Background

In this section we formally define the main concepts that we will use in the rest of the paper. A correlation set, c-set for short, can be seen as a key that can be used to retrieve a session. For our purposes a c-set can be seen as a set of variables names (in BPEL these correspond to c-set proprieties) that can assume values in a domain. To simplify the notation we assume that the variables of a c-set can assume values in the domain D defined as the set of strings on a given signature.

Definition 1 (c-set). *Given a service S, a correlation set for S is a finite set of variables names. When these variables are defined their values uniquely identify a session of S.*

Sessions may define the variables of a c-set. The definition of variables belonging to a c-set is captured with the following definition.

Definition 2 (c-instance). *Given a c-set c we say that a c-instance for c is a total function that maps every variable of c to a value in D.*
We will say that a session s has a c-instance φ if for every variable v in c the variable v has been assigned and its value is $\varphi(v)$.

Services, especially those having multi-party sessions, may need more than one c-set because the users may need to use different keys to identify a session. These services, also known as multi correlation services, do not require to have a c-instance for every c-set. However since c-sets are used to identify a session we require that a session must have at least a c-instance. Moreover we do not allow the starting of a session having the same c-instance of another existing session.

Every message that is exchanged will contain some arguments associated to a c-set. Usually these arguments are called correlation tokens or correlation values and are used to find the recipient of the message. BPEL and other service engines allow the use of potentially one correlation token (c-token for short) for every c-set of the service. For example a multi-party session can be initialized submitting a message having as correlation tokens the values for all the c-sets of the service. In this work instead we will consider messages having only one c-token. This restriction is however insignificant since the behaviour that is caused by the exchange of messages with more than one c-token can be easily simulated in our framework. This is due to the fact that differently from BPEL we do not need the exchange of a message to change the value of a correlation variable.

Formally we can define a c-token in the following way.

Definition 3 (c-token). *Given a message m a c-token is a pair (c, φ) where*

- *c is a c-set containing the variables used to specify the message recipients*

- *if m is a normal message then φ is a total function that maps a variable of c into a value in D*
- *if m is a broadcast message then φ is a partial function that maps a variable of c into a value in D. Moreover φ is not total.*

For instance the service for buying goods has only one c-set $c = \{name, surname\}$ and the c-instance of John's session is the function φ s.t. $\varphi(name) = John$ and $\varphi(surname) = Smith$. The message $buy(John, Smith, 1)$ has instead as c-token the couple (c, φ). If we want to send a message m to every person named John for wishing him a happy name day we can use a broadcast message whose c-token will be the couple (c, φ') where $\varphi'(name) = John$ and $\varphi(surname)$ is not defined.

As it can be seen in the previous definition the introduction of the broadcast primitive allows the user to not define all the variables of a c-set. Normal messages, like c-instances, need to define all the variables of a c-set because they need to identify their (unique) target session. On the other hand, broadcast messages can specify only a part of the key, indeed their target can be a set of sessions. Note that, in case of multi correlation services, the c-token definition does not allow to consider part of two different keys to determine the targets of a broadcast message. We do not allow this possibility since we haven't find a significant example that justifies this increased power. However we could easily extend our framework to treat also this case. Now we can formally define when a message correlates with a session. Intuitively a message correlates with a session when the values of the correlation token match the c-instance of a session. In the following $\varphi_m(v) \uparrow$ denotes that φ_m is not defined in v.

Definition 4 (Correlation). *Given a service S, a session s and a message m with c-token (c_m, φ_m) we will say that s correlates with m iff s has a c-instance φ for the c-set c_m and $\forall v \in c_m.\ \varphi_m(v) = \varphi(v) \vee \varphi_m(v) \uparrow$.*

3 The idea

As we have discussed above the current mechanisms for assigning a message to the correct session does not support the possibility of identifying a set of sessions. A naive implementation for the support of broadcast messages would use an associative array for every c-set variable. However, if this solution is used, for finding the targets of a broadcast message we have to compute a set intersection whose complexity depends on the number of sessions. Another naive solution is using an associative arrays for every subsets of correlation variables that can be used in a broadcast message. If we consider a c-set with n variables this means that for the support of the broadcast primitive we could have $2^n - 1$ associative arrays, since with n variables we can use up to $2^n - 1$ different kind of broadcast messages (one for every subset of the c-set variables). Our key idea in order to improve on this is to use radix trees to memorize the c-instances of all the sessions and therefore for routing messages to the correct session. In this

section we will explain intuitively the idea, while its formalization and complexity analysis are contained in the next sections.

A trie, or a prefix tree, is an ordered tree for storing strings, in which there is one node for every common prefix. Edges are labeled with characters, while the strings are stored in extra leaf nodes. Tries are extremely useful for constructing associative arrays with keys that can be expressed as strings, since the time complexity of retrieving the element with a given key is linear time in the length of the key. In fact, looking up for a key of length k consists in following a path in the trie, from the root to a leaf, guided by the characters in the key. A radix tree (or Patricia tree, [10]) is essentially a compact representation of a trie in which any node that has no siblings is merged with its parent (so, each internal node has at least two children). Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters. This makes radix tree more efficient than tries for storing sets of strings (keys) that share long prefixes. The operations of lookup (to determine whether a string is in the set represented by a radix tree), insert (of a string in the tree), and delete (of a string from the tree) have all worst case complexity of $O(l)$, where l is the maximal length of the strings in the set.

Intuitively our idea is to use radix trees to map incoming messages to sessions, by using the values of the c-set variables as keys. In other words, the session pointers can be seen as elements stored in an associative array, while the values of the variables of the c-sets, conveniently organized as strings, are the keys. Our radix trees implements such a structure by memorizing the values of the c-set variables which appear in the existing sessions. In particular, since every broadcast message can define only part of the c-set variables, to be able to process every message we could use a radix tree for every subset of the c-set variables. This however is not an optimal solution. For example if a service has two c-set variables *name* and *surname* we could receive the following kind of messages

1. broadcast messages s.t. their c-tokens do not define any variable
2. broadcast messages s.t. their c-tokens define only the field *name*
3. broadcast messages s.t. their c-tokens define only the field *surname*
4. normal messages s.t. their c-tokens define both *name* and *surname*

With the naive approach we need to use 4 associative arrays (one for every message type). Using radix trees is however possible to use a unique radix tree for 1^{st}, 2^{nd} and 4^{th} types since the c-tokens of the 1^{st} and 2^{nd} kind of messages can be considered as prefix of the c-tokens of the 4^{th} type of messages. For the message of the 3^{th} type instead we have to use a different radix tree, since in this case the c-tokens are not a prefix of those for the 4^{th} type of messages. So it is sufficient to use two radix trees to cover all the possible cases.

To better explain the idea let us consider some more examples. In the following we use a special character, denoted by # and not used elsewhere, to denote in a string the termination of the values of c-set variables.

We first consider a unique c-set variable with only one field: *name*. When there exist no session for such a variable we have a radix tree consisting of the

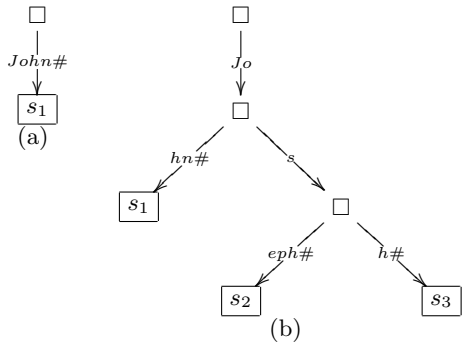


Fig. 1. Example of radix trees

only root (recall that in radix trees the root is associated with the empty string). We represent such a radix tree as a $\square$. If now a session s_1 is created which is identified by the value *John* for the c-set variable *name* then the radix tree became as the one depicted in Figure 1(a). The value *John* allows to reach s_1 by an (obvious) lookup in the tree.

Next assume that two more sessions are created: a session s_2, which is identified by the value *Joseph* for the variable *name* and a session s_3 which is identified by *Josh*. The radix tree we obtain is the one depicted in Figure 1(b). Notice that the longest common prefixes of the three key values are associated to edges of the tree. When a message arrives, the value that it carries for the *name* variable allows one to select a root-leaf path in the tree, so reaching the correct session.

Assume now that our correlation set is composed by the two variables *name* and *surname* and consider four sessions $s_1 - s_4$ identified as follows by the values of the c-set variables:

$$s_1 : name = John, surname = Smith;\ s_2 : name = John, surname = Smirne$$
$$s_3 : name = Josh, surname = Smith;\ s_4 : name = John, surname = Smithson$$

Correspondingly we have the radix tree depicted in Figure 2(a). In this case, as mentioned before, we need more that one radix tree to store the values of c-sets variables of the sessions. This because in a broadcast message the value of some c-set variables could be not specified. For example, in the case above, let us consider a broadcast message which contains the token *Smith* for *surname* and no token for *name*. If we have only a radix tree like the one depicted in Figure 2(a) we can not find with a lookup which session correlate with it. This is due to the fact that the first part of the key of the radix tree is the value of the variable *name*. Hence we need an additional radix tree like the one depicted in Figure 2(b) that can be used to retrieve sessions for messages that do not define the variable *name*.

It is easy to see that these two radix trees allow to cover all the possible cases. First consider what happens if we receive a message m where $name = John$ and $surname = Smith$, hence we consider the string *John#Smith#*. In this case, by using the 2(a) radix tree, we see that the message m will be assigned to s_1, since this is the session which correlates with m. However, note that this

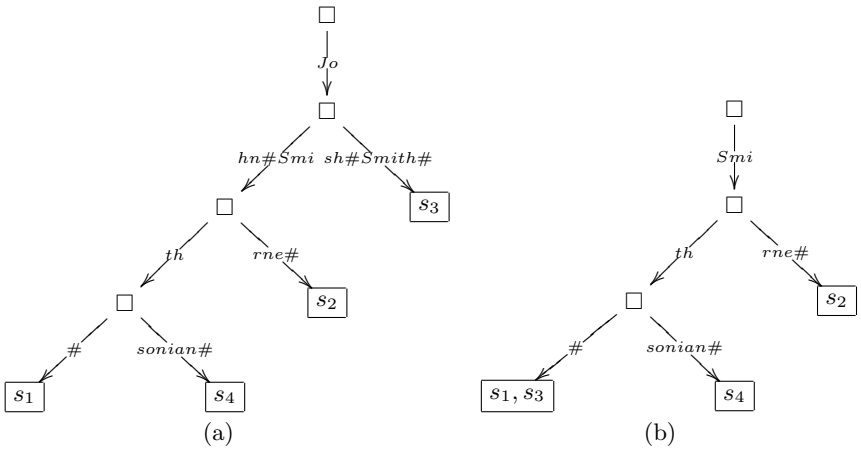


Fig. 2. Example or radix trees for c-set with 2 variables

first tree covers also the case in which no value for surname is provided by the message, hence we do not need a further radix tree to keep only the sessions that define only the variable *name*. For example, if we receive a message m with $name = John$, that is we consider the string $John\#$, then the 2(a) radix tree shows that m correlates to the sessions s_1, s_2, s_4.

On the other hand, if we receive a broadcast message m' where *name* is not defined and $surname = Smith$ we will use the 2(b) radix tree (with the string $Smith\#$) to find that the session correlating with m' are s_1, s_3.

4 Building the radix trees

As previously discussed, with our approach every c-set of the service has a group of radix trees that can be used for checking the correlation of a message to a session. We have also shown that, if we assume that the c-set has n variables, one does not need to consider 2^n different radix trees, because a radix tree for a sequence of variables cover also all the cases given by the prefixes of such a sequence.

In this section we provide an algorithm that, given a c-set with n variables, in the worst case constructs a set containing $\binom{n}{\lceil n/2 \rceil} (= \frac{n!}{\lceil n/2 \rceil! \ \lfloor n/2 \rfloor!})$ radix trees. In the next section we will prove that such set allow us to route all the possible messages to a service. We also prove that this set is minimal, in the sense that any other set of radix trees which allow to route correctly all the messages has at least the same cardinality. So our algorithm cannot be improved w.r.t. the number of radix trees generated.

In the following we assume that the c-set c has n variables and the set V contains all and only these variables. We denote by seq_i a sequence $x_1, \ldots, x_{h_i}$ of variables of c. Given a list of sequences of variables $seq_1, \ldots, seq_m$ such that seq_i

is a prefix of seq_{i+1}, for $i \in [1, m-1]$, we use the notation $RT(seq_1, \ldots, seq_m)$ to indicate any radix tree whose keys are strings of the form $d_1\# \ldots \#d_{h_i}\#$ where $d_j = \varphi(x_j)$, for $j \in [1, h_i]$, and for some c-set-instance φ. In other words, $RT(seq_1, \ldots, seq_m)$ is a kind of schema which can be instantiated by considering the values of the variables for one specific sequence seq_i, with $i \in [1, m]$ (and using # as separator of values), to obtain a specific concrete radix tree. As previously discussed, a radix tree (described by) $RT(seq_1, \ldots, seq_m)$ allows us to check the existence of a session defining all the variables in one of the sequences seq_i. For example the radix tree in Figure 2(a) can be denoted by $RT(\langle\rangle, \langle name\rangle, \langle name, surname\rangle)$ while the radix tree 2(b) is denoted by $RT(\langle surname\rangle)$ [5]. By using this notation our problem can be stated as follows: we need to find the minimum number h of radix trees schemas $RT_1(seq_{1,1}, \ldots seq_{1,l_1})$, $\ldots$, $RT_h(seq_{h,1}, \ldots seq_{h,l_h})$ such that, for each set $X \subseteq V$, there exists a sequence $seq_{k,o}$ that contains all and only the variables in X.

We find convenient to formulate this problem in terms of a graph representation. Indeed, given a set of variables V, we can create a labeled direct graph $G(V)$ where:

- the nodes are (labeled by) elements in $\mathcal{P}(V)$. Intuitively we will consider all the set of variables that can be defined by a c-token;
- there is an arc from u to v if $u \subset v$;
- the arc (u, v) is labeled with the variables $v \setminus u$ (where $\setminus$ denotes set difference).

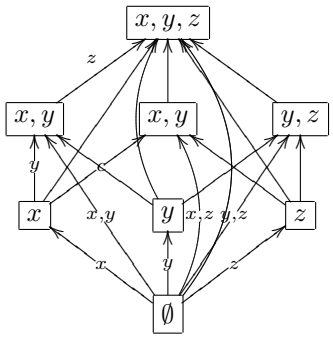


Fig. 3. Example of the graph obtained for with three variables: x, y, z (note that for convenience only few arc labels are reported)

For example, in Figure 3 we see the graph constructed by considering the three variables x, y and z where we can receive all the possible 7 broadcast messages. A path on this graph corresponds to a radix tree schema (see definition 5). Hence, with this graph representation our problem can be stated as follows: we have to find the minimum number of paths that cover all the nodes of the graph where, as usual, we say that a path $u_1 \xrightarrow{x_1} u_2 \xrightarrow{x_2} \ldots \xrightarrow{x_n} u_{n+1}$ covers the nodes $u_1, \ldots, u_{n+1}$.

[5] Note that the order of the cset variables is important and therefore for instance $RT(\langle name, surname\rangle) \neq RT(\langle surname, name\rangle)$

The algorithm that produces this minimum number of paths is Algorithm 1 and its intuition is the following. Consider the graph $G(V)$ associated to a c-set V, as explained above. We first partition all the nodes of $G(V)$ into levels according to the number of variables of the nodes, so level i contains all the nodes that have exactly i variables. Then starting from the lowest levels (i.e. level 0 and 1) we consider two next levels at a time, say level i and $i+1$. These two levels are seen as a bipartite graph where the nodes of each level form an independent set. We then use a maximum bipartite matching algorithm for selecting a set of arcs between the nodes of these two levels. Next we repeat the same procedure with levels $i+1$ and $i+2$, and we continue until we reach the level n. At this point we take the graph $G'(V)$ obtained by considering all the nodes in the original graph $G(V)$ and only the edges which have been selected by the matching algorithm. As we prove in the next section, the maximal paths[6] on the graph $G'(V)$ form a minimum set of paths covering all the nodes of P.

Before providing the algorithm we need to introduce some notation. We assume that each node is (labeled by) an element of $\mathcal{P}(V)$ ($n = |V|$), as mentioned above and we denote by $level_V(i)$ the set of nodes in the i-th level, i.e. the set of elements in $\mathcal{P}(V)$ which have cardinality i. Moreover $graph(A, B)$ denotes the bipartite direct graph $(A \cup B, E)$ where $(u, v) \in E$ iff $u \subset v$. Finally $maximal\_matching(G)$ is one of the maximal matchings of the bipartite graph G chosen in a non deterministically way. Algorithm 1 takes as input the set $P \subseteq \mathcal{P}(V)$ and returns the graph containing a minimum set of paths covering all the nodes of P. Once we have obtained a graph by using the Algorithm 1 it

Algorithm 1 $radix\_trees(P)$

```
 1: i = 0
 2: V = level_P(i)
 3: M = ∅
 4: while (i < n) do
 5:    i = i + 1
 6:    V' = level_P(i)
 7:    G = graph(V, V')
 8:    M' = maximal_matching(G)
 9:    V = V − {v | (v, x) is an edge in M', for some x}
10:    V = V ∪ V'
11:    M = M ∪ M'
12: end while
13: return (P, M)
```

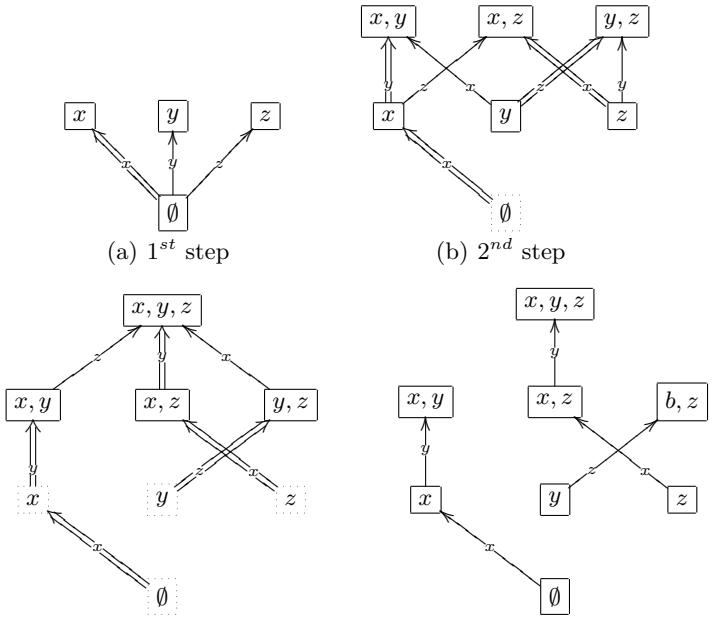
is possible to compute the radix trees by simply finding all the maximal paths, as shown below.

Definition 5. *Given $P \subseteq \mathcal{P}(V)$ we say that a radix tree schema $RT(u'_1, u'_2 \ldots, u'_m)$ is produced by the algorithm $radix\_tree(P)$ if $u_1 \xrightarrow{x_1} u_2 \xrightarrow{x_2} \ldots \xrightarrow{x_m} u_{m+1}$ is a maximal path in the graph $G = radix\_tree(P)$ and*

[6] A maximal path is a path that can not be a proper part of another path.

– u'_i *is a sequence of all the variables in the set* u_i*, for each* $i \in [1, m]$*;*
– u'_i *is a prefix of* u'_{i+1}*, for each* $i \in [1, m-1]$.

We now consider an example of application of the previous algorithm to the graph in Figure 3. In Figure 4 we have reported the three steps denoting by $\Rightarrow$ the arcs selected by the maximal matching algorithm (i.e. arcs in M) while $\rightarrow$ indicates the arcs considered by the maximal matching algorithm (i.e. arcs in G, line 7). The nodes in frame are the nodes that are used for computing the maximal matching (i.e. the nodes in V and in $level_P(i)$), while nodes in dotted frame are the nodes already processed (not considered by the matching algorithm and deleted from V, line 9).

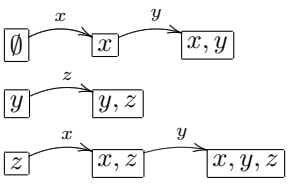


(a) 1^{st} step (b) 2^{nd} step

(c) 3^{rd} step (d) Final graph

Fig. 4. Example of execution of Algorithm 1 with 3 variables

From the final graph (Figure 4(d)) we can compute the radix trees schemas by taking the maximal paths:



The first path corresponds to the radix tree schema $RT(\langle\rangle, \langle x\rangle, \langle x, y\rangle)$ while the other two corresponds to $RT(\langle y\rangle, \langle y, z\rangle)$ and $RT(\langle z\rangle, \langle z, x\rangle, \langle z, x, y\rangle)$, respectively.

4.1 Using radix trees

Once we have created the radix tree schemas by using our algorithm, we need some operations for inserting and removing values from them, thus creating the concrete radix trees to be used for correlating messages and sessions. Moreover

we need to define a lookup operation, that, given a message, allows us to use the (concrete) radix tree to find all the correlating sessions. To this aim we first introduce the three operations described below. Here and in the following, unless differently specified, with "radix tree" we mean a concrete radix tree, containing values for keys and whose leafs contain (pointers to) sessions. Moreover we assume w.l.o.g. that the service has a unique c-set and therefore only one group of radix trees. If the service has more than one c-set the following considerations should be applied to every c-set.

- $RT.add(s)$ is the operation for adding to the radix tree RT the session s;
- $RT.del(s)$ is the dual operation that deletes the session s in RT;
- $RT.find(m)$ returns all the sessions which correlate with m. If no sessions in RT correlates with m then the *null* pointer is returned.

Assuming that RT belongs to the radix tree schema $RT(seq_1, \ldots, seq_k)$, when $RT.add(s)$ is invoked s is added to the radix tree RT using as key the string $\varphi_s(x_1)\# \ldots \#\varphi_s(x_l)\#$ where $\langle x_1, \ldots, x_l \rangle = seq_k$ and φ_s is the c-instance for s. In a similar way $RT.del(s)$ deletes from RT the session pointer to s.

If $\langle x_1, \ldots, x_l \rangle$ is the sequence of all the variable defined by the c-token φ of a message m, the operation $RT.find(m)$ can be applied iff there exists a sequence $seq_i = \langle x_1, \ldots, x_l \rangle$. In this case this operation returns all the sessions whose keys have as prefix the string $\varphi(x_1)\# \ldots \#\varphi(x_l)\#$.

Using these basic operation we can now define the operations which manage the set of radix trees produced by our algorithm . More precisely, we assume that the set of radix tree schemas produced by the algorithm has been instantiated to a set of (concrete) radix trees. Then this set is managed by the following three operations: $find\_session(m)$ (for finding a session that correlates with a message m); $add\_session(s)$ (for adding the session s); $del\_session(s)$ (for deleting a session s). The definition of the $add\_session(s)$ and $del\_session(s)$ is obvious since the only thing to do is to execute $RT.add(s)$ and $RT.del(s)$ for every radix tree RT. The $find\_session(m)$ instead first have to select a specific RT based on the variables defined by the c-token of m and later return the $RT.find(m)$ result.

5 Correctness and complexity analysis

In this section we prove the correctness of Algorithm 1 and we discuss the complexity of correlation mechanism based on it. In particular, we show that it produces the minimal number of radix trees needed to guarantee correctness. In the following, as usual, we assume that V is the set of variables of a c-set and that $n = |V|$.

First of all, we show that Algorithm 1 produces a number of radix trees much smaller than 2^n. With a slight abuse of notation, when no ambiguity arise, we indicate by $radix\_trees(P)$ both the graph produced by the algorithm, with input P, the radix tree schemas obtained from this graph according to Definition 5, and the concrete radix tree obtained from the schemas as described at the end of previous section. All the proofs of the theorems are reported in [7].

Theorem 1. *If $W \subseteq \mathcal{P}(V)$ the result of radix_trees(W) is a graph containing at most $\binom{n}{\lceil n/2 \rceil}$ maximal paths. Hence the algorithm produces at most $\binom{n}{\lceil n/2 \rceil}$ radix trees schemas.*

Next we show that the algorithm is correct, that is, the number of radix trees produced is sufficient to check correlation.

Theorem 2. *Let m be a message and $V_1, \ldots, V_k$ be all the subsets of c-set variables that are defined by all the possible c-tokens. Then there exists a radix tree schema produced by radix_trees$(\{V_1, \ldots, V_k\})$ which allows us to check if the message correlates with a session.*

Finally we show that the number of radix trees produced by the algorithm is the minimal one which guarantees correctness.

Theorem 3. *The graph produced by radix_trees(P) contains the minimal number of maximal paths covering all the nodes in P.*

As an obvious consequence of previous theorem we obtain that if we consider less radix trees than those produced by Algorithm 1 we cannot establish correctly correlation for some kind of messages. Thus our algorithm cannot be improved with respect to the number of radix trees that one can use to solve this problem.

The complexity of Algorithm 1 is polynomial on the size of P. As for the complexity of the operations described in Section 4.1, assuming that l is the maximum length of a c-set value and k is the number of the sessions that correlate with a message m, the (time) complexity of $find\_session(m)$, is $O(n+knl) = O(knl)$. For normal (i.e. non broadcast) messages the complexity of $find\_session(m)$ reduces to $O(nl)$. On the other hand, the (time) complexity of $add\_session(s)$ and $del\_session(s)$ is $O(\binom{n}{\lceil n/2 \rceil} l)$ (for more details see [7]). We would like to underline that, in practice, the number of the c-set variables which are used is very small (less or equal to 5) so, in practice, the complexity of our operations is constant.

6 Conclusions and future work

We have proposed a data structure, based on radix trees, for managing a correlation mechanism which supports also a broadcast communication in the context of languages for service oriented computing. We have also described an algorithm that computes the minimal number of radix trees required for handling correctly every normal and broadcast message. The complexity of the correlation operation is constant for normal messages, and linearly dependent with respect to the number of targets for broadcast messages. The operations of session creation and termination have a complexity that depends on the number of different types of broadcast messages. In the worst case (i.e. when an exponential number of broadcast messages is used) it is exponential. The worst case scenario is however impossible in practice, since real scenarios use few types of broadcast messages. For this reason the complexity of session creation and termination have in practice a constant complexity.

The major drawback of our approach is memory consumption: having more than one radix tree means that we require more memory to store the correlation values. For services that use huge data as correlation values memory consumption could be problematic. Nevertheless, we believe that in practice this is not an issue, since correlation values should be small for minimizing the cost of the message exchange over the network. If a service uses huge data as correlation values then we argue that it is worth considering the introduction of a new shorter key that can be used as a new correlation variable.

We are currently implementing the data structure and the algorithm in the JOLIE language interpreter. With this new implementation hopefully we will be able to provide a faster mechanism for the assignment of messages to session.

We are currently extending our work to support a property-based correlation mechanism (see [2]) where also such operators as $>, <, \vee$ can be used for the assignment of messages and therefore the analogous of range queries in databases arise. We think that radix trees could be very useful in this context, since these data structures allow to manage range queries in a very natural way.

References

1. *Web Services Business Process Execution Language Version 2.0*. Available at `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.
2. Alistair P. Barros, Gero Decker, Marlon Dumas, and Franz Weber. Correlation patterns in service-oriented architectures. In *FASE*, pages 245–259, 2007.
3. Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.
4. Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.
5. Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A calculus for service oriented computing. In *ICSOC*, 2006.
6. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *ESOP*, pages 33–47, 2007.
7. Jacopo Mauro, Maurizio Gabbrielli, Claudio Guidi, and Fabrizio Montesi. An efficient management of correlation sets with broadcast. Technical report, 2011. Available at `www.cs.unibo.it/~jmauro/papers/tech_report_coordination_2011`.
8. Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. JOLIE: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.
9. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with jolie. In *ECOWS*, pages 13–22, 2007.
10. Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
11. Gero Mühl. Generic constraints for content-based publish/subscribe. In *CoopIS*, pages 211–225, 2001.